序列dp

2019年4月19日 14:55

一. Longest Increasing Subsequence最长上升子序列LIS

1. 模板：单调递增子序列的最大长度（子序列原下标不用连续，但要递增）
   i. 阶段i：对下标的顺序遍历
   ii. 状态ans[i]：以下标i结尾的上升子序列
   iii. 转移方程：if(a[i]>a[j]) ans[i]=max(a[i],a[j]+1);
   iv. 初值都是0
   v. 答案：ans数组中的最大值
2. 拦截导弹
   i. 题意：一个拦截系统只能依次拦一个长度不升序列的各导弹，求最强的一个系统能拦几个导弹，以及最少要几个系统
   ii. 分析：最多导弹数即最长不升子序列数，最少系统数即最少不升子序列数
   iii. Dilworth定理的应用：**最少不升子序列数=最长上升子序列长度**（和偏序集的反链有关？）
   iv. 可以用d[i]记录i开始的最长不升序列长度，u[i]记上升序列长度
   v. 阶段i一定是要逆序遍历的，j的决策倒是无所谓顺逆
   vi. 因为懒得赋初值1，所以利用静态数据初值0，在最后才+1
   vii. ans和ans2似乎在u和d更完了再一次找更快，但懒得多写个循环了

```
rof__(i,n-1,0)
    for_(j,i+1,n){
        if(h[i]>=h[j])
            d[i]=max(d[i], d[j]+1),
            ans=max(ans, d[i]);
        if(h[i]<h[j])
            u[i]=max(u[i], u[j]+1),
            ans2=max(ans2, u[i]);}
cout<<ans+1<<" "<<ans2+1;
```

3. 单调序列二分法，洛谷P1020
   i. 如果是单调的，就加入栈，不单调，就二分替换“栈”里比他差的元素
   ii. 上升栈a里是大于等于，不升栈na里必须严格小于

```
inline int* ge(int *a, int *b, int x) {  // 第一个大于等于
    return lower_bound(a, b, x);
}
inline int* lt(int *a, int *b, int x) {  // 第一个严格小于
    return upper_bound(a, b, x, greater<int>());
}

inline void solve() {
    int n=0;
    while(~scanf("%d",x+n)) ++n;
    na[nat]=a[at]=x[0];
    for(int i=1; i<n; ++i) {
        if(x[i] > a[at]) a[++at] = x[i];  // 上升的，尾插
        else *ge(a, a+at, x[i]) = x[i];  // 不升的，替换比他大的
```

```
            if(x[i] <= na[nat]) na[++nat] = x[i];  // 不升的，尾插
            else *lt(na, na+nat, x[i]) = x[i];  // 上升的，替换不比他大的
        }
        printf("%d\n%d",++nat,++at);
        // 最长不升序列长度，最长上升序列长度(=最少不升序列数)
    }
```
来自 <https://tool.oschina.net/highlight>

4. 树状数组法：维护值域上从小到大和从大到小里的dp最大值

二. Longest Common Subsequence最长公共子序列LCS

1. 问题：既为a的子序列，又为b的子序列的子序列的最大长度
2. 阶段ij：顺序遍历到a的下标i和b的下标j（设长度分别为N，M）
3. 状态ans[i][j]：遍历到i和j时找到的LCS长度（符合最优子结构性质）
4. 转移方程1：**ans[i][j]=max(ans[i-1][j],ans[i][j-1])**
5. 转移方程2：**if(a[i]==b[j]) ans[i][j]=max(ans[i][j],ans[i-1][j-1]+1)**
6. 初值：每一列ans[0][j]为0，每一行ans[i][0]为0，其他的可以不赋值
7. 答案：ans[N-1][M-1]（ij各循环到N-1，M-1时）

```
vector<vector<int>> d(len1, vector<int>(len2));
d[0][0] = text1[0] == text2[0];
for(int j=1; j<len2; ++j)
    d[0][j] = d[0][j-1] || text1[0]==text2[j];
for(int i=1; i<len1; ++i)
    d[i][0] = d[i-1][0] || text1[i]==text2[0];
for(int i=1; i<len1; ++i)
    for(int j=1; j<len2; ++j)
        if(text1[i] == text2[j])
            d[i][j] = d[i-1][j-1] + 1;
        else
            d[i][j] = max(d[i-1][j], d[i][j-1]);
return d[len1-1][len2-1];
```

8. 注意：i-1和j-1可能越界，而用i+1和j+1又会使首字符的相同与否不被判断到，最简单的解决方法是可以空出dp数组的下标0
9. 优化：先循环i再循环j的话，可以只开j所在纬度的数组，这样更新d[i][j]时d[i-1][j]和d[i][j-1]恰存在d[j]和d[j-1]中；保存好d[i-1][j-1]的话，即可转移，但要注意d[i-1][j-1]在每次i循环都要重新清零

```
vector<int> d(len2+1);
for(int djlast=0,i=1; i<=len1; ++i,djlast=0)
    for(int djnow,j=1; j<=len2; ++j) {
        djnow = d[j];
        if(text1[i-1] == text2[j-1])
            d[j] = djlast + 1;
        else
            d[j] = max(d[j], d[j-1]);
        djlast = djnow;
    }
return d[len2];
```

-
- 序列和

三. 最大m个不相交序列和（指拆成最多m个子串的和的最大值）

1. 引：用d[j]存e[j]结尾的子串和，允许拆成两个子串的话，易知从前i-1项作结尾的

子串和的最大值转移过来很可能是最优解（此处转移指加上e[j]）

2. 另：如果e[j-1]很大，而且d[j-1]保存了前j-2项的单个子串最大和+e[j-1]的话，也许这个值会比单个子串和更大，而因为e[j]紧接着e[j-1]，跟在它后面也只是两个子串，是符合题意的
3. 允许2个子串的转移方程是d[2][j]=max(d[2][j-1]+max{d[2-1][j-1]})+e[j]
4. 转：有m个子串的话其实也可以类似上述1转2的方式从i-1转至i，即转移方程**d[i][j]=max(d[i][j-1]+ max{d[i-1][j-1]})+e[j]**
5. 另：易知允许i个子串与否对j<i时的e[j]没什么影响，j从i开始循环即可
6. m范围较大时复杂度也过大，注意到i只从i-1转，可以用一个单个数组
7. 又注意到求上一轮的前j项最大值可以在上一轮顺便求出来，注意顺序即可
8.
```
int e[1000005];
int d[1000005];          //d[j]=e[j]结尾的最大字段和
int M[1000005];          //M[j]=上一行的max{d[k]}当k<=j
while(~scanf("%d%d",&m,&n)){
        for__(i,1,n)
                scanf("%d",e+i);
        memset(d, 0, sizeof(d));
        memset(M, 0, sizeof(M));
        for__(i,1,m){    //每更新一轮循环，就允许d[j]存储多一个子段后的值
                ans= 1<<31;      //负无穷，防止上一行M[n]这一行M[i]
                for__(j,i,n){
                        d[j]= max(d[j-1], M[j-1])+ e[j];
                                //选d[j-1]相当于只允许e[j]紧接在上一个
                                //选M[j-1]是在前者基础上，允许跳过e[j-1],e[j-2]......
                        M[j-1]= ans;
                                //上一行代码后再更新M[j-1]=这一轮前j个d的最大值
                                //方便下一行计算允许分成i+1段时
                        ans= max(ans, d[j]);}}
                                //上一行代码后再更新ans=这一轮前j个d的最大值
        cout<<ans<<endl;}
```

四. 回文列

1. 回文子序列数（在原串的相对位置不变，但不要求再原串连续）
    i. d[i][j]存储从i到j的所有回文数
    ii. 思路：s[i]和s[j]不相等时，显然不可能同时以他俩为首末形成回文，所以转移=所有不用头+所有不用尾，再容斥-所有不用头又不用尾
    iii. 思路：s[i]和s[j]相等时，可能同时以他俩为首末了，转移=上一条转移+不用头又不用尾的情况下以他俩为首尾+空串的情况下以他俩为首尾

```
rof_(i,len-1,-1){         //为了从i+1转移到i
        d[i][i]=1;        //为防止j-1越界，i==j的情况单独赋值
        for_(j,i+1,len)
                if(s[i]==s[j])
                        d[i][j]=d[i+1][j]+d[i][j-1]+1;
                else
                        d[i][j]=d[i+1][j]+d[i][j-1]-d[i+1][j-1]; }
```

```
cout<<d[0][len-1];
```